CS2102 Finals
AY23/24 SEM 2
github/SelwynAng

1 Relational Model

1.1 DBMS

- **Transactions:** Finite sequence of operations & constitutes smallest logical unit of work from application perspective
- **Properties of Transactions:**
  1. Atomicity: Either all effects of Transaction are reflected in database or none
  2. Consistency: Transaction guarantees to yield correct state of database
  3. Isolation: Transaction is isolated from effects of concurrent Transactions
  4. Durability: After commit of Transaction, effects are permanent
- **Equivalent Transaction:** 2 executions are equivalent if they have same effect on database
- **Serialisability:** Concurrent execution of a set of transactions is serialisable if execution is equivalent to some serial execution of the same set of Transactions

1.2 Relational Model

Term	Description
Attribute	column of a table
Domain	possible values
Attribute Value	element of a domain
Relation Schema	set of attributes + constraints + relation name
Tuple	row of a table
Relation	set of tuples
Cardinality	number of rows
Database Schema	set of relation schema
Database	set of relations

Relational Schema: Defines a relation, specifies attributes (columns), data constraints, name

- $R(A_1, A_2, \ldots A_3)$: relation schema with name R and n attributes $A_1, A_2, \ldots, A_n$
- Eg. `Employees(id:INT, name:TEXT, dob:TEXT)`

Domain: Set of atomic values, including NULL

- $dom(A_i)$ = set of possible values for A_i
- $\forall$ value v of attribute A_i, $v \in \{dom(A_i \cup \{null\})\}$

1.3 Keys

- **Superkey:** Subset of attributes that UNIQUELY IDENTIFIES a tuple in a relation
- **Key:** Superkey that is also MINIMAL (No proper subset of key is a superkey)
- Properties of Superkeys & Keys:
  1. If (A,B,C) is DEFINITELY superkey → (A,B,C,D) is superkey
  2. If (A,B,C) is DEFINITELY key → (A,B,C) is superkey
  3. If (A,B) is DEFINITELY superkey → (A,B,C) is NOT a key
  4. If (A,B) is DEFINITELY key → possible (B,C,D) is also key
  5. Every relation has ≥ 1 superkey
- **Candidate Key:** Set of all keys of a given relation
- **Primary Key:** A selected candidate key (attributes CANNOT be NULL, is underlined in schema notation)
- **Foreign Key:** Subset of attributes of relation R_1 that refers to Primary Key of relation R_2
  - R_1 is referencing relation, R_2 is referenced relation
  - $R.sid \rightarrow S.id$: $R.sid$ is a FK referencing PK id in S
  - FK in R_1 must appear as PK in R_2 OR be NULL/tuple containing at least 1 NULL value
  - A referencing relation can be a referenced relation for different foreign key | Referencing relation & referenced relation can be same relation

2 SQL

2.1 Three-valued Logic / Handing NULLs

- **Logical Operations**

Conjunction

c_1 AND c_2		c_1 False	c_1 NULL	c_1 True
c_2	False	False	False	False
	NULL	False	NULL	NULL
	True	False	NULL	True

Disjunction

c_1 OR c_2		c_1 False	c_1 NULL	c_1 True
c_2	False	False	NULL	True
	NULL	NULL	NULL	True
	True	True	True	True

- **Implication:** $C_1 \rightarrow C_2 == (\sim C_1) \vee C_2$
- $\sim NULL == NULL$
- **Relational Operations:** Any relational operations with $NULL$ produced $NULL$ values (Eg. $\leq, =, !=$)
- **Arithmetic Operations:** Any arithmetic operations with $NULL$ produces $NULL$ values

SQL has additional operations to treat NULL as values.

v_1	v_2	v_1 IS NOT DISTINCT FROM v_2	v_1 IS DISTINCT FROM v_2	v_1 IS NULL	v_1 IS NOT NULL
NULL	NULL	True	False	True	False
NULL	v_2	False	True	True	False
v_1	NULL	False	True	False	True
v_1	v_2	$v_1 = v_2$	$v_1 <> v_2$	False	True

- **IS NULL vs = NULL (in WHERE clause):** IS NULL will select NULL values | = NULL will not select any NULL values (NULL = NULL → NULL, nothing will be selected by Principle of Acceptance)
- **DISTINCT keyword (in SELECT clause):** DISTINCT checks for distinct rows using IS DISTINCT FROM (Duplicate NULL values are removed)
- **Empty & NULL Semantics in Aggregate functions**

Empty Semantics

Let $R1$ be an empty relation with attribute A.

Query	Result
SELECT MIN(A) FROM R1;	NULL
SELECT MAX(A) FROM R1;	NULL
SELECT SUM(A) FROM R1;	NULL
SELECT AVG(A) FROM R1;	NULL
SELECT COUNT(A) FROM R1;	0
SELECT COUNT(*) FROM R1;	0

NULL Semantics

Let $R2$ be a non-empty relation with n-rows and with attribute B that only has NULL values.

Query	Result
SELECT MIN(B) FROM R2;	NULL
SELECT MAX(B) FROM R2;	NULL
SELECT SUM(B) FROM R2;	NULL
SELECT AVG(B) FROM R2;	NULL
SELECT COUNT(B) FROM R2;	0
SELECT COUNT(*) FROM R2;	n

2.2 Integrity Constraints

- **Principle of Rejection:** Integrity Constraints follow POR (Rejects insertion if condition evaluates to FALSE, Insertion is still done if condition is NULL)

1. NOT NULL: Rejects insertion if value at specified column is NULL (Condition: IS NOT NULL)
2. UNIQUE: Rejects insertion if there are other rows where values are equal (Condition: $x.A_i <> y.A_i$) | Can have multiple NULL values since NULL <> NULL == NULL
3. Primary Key: Equivalent to UNIQUE & NOT NULL | If one of the attributes is NULL, entire tuple is rejected
4. Foreign Key: Rejects insertion if tuple does not exist in referenced relation AND is NOT NULL
   - NO ACTION: Rejects delete/update if it violates constraints
   - CASCADE: Propagates delete/update to referencing tuples
   - SET DEFAULT: Updates FK of referencing tuples to default value
   - SET NULL: Updates FK of referencing tuples to NULL value
5. CHECK: Rejects insertion if condition is FALSE

2.3 Deferrable Constraints

- **Default Behaviour:** Constraints are checked immediately at end of SQL statement → Violation of 1 of the statements will cause whole transaction to roll black
- **Benefit:** Allow cyclic FK constraints

1. NOT DEFERRABLE: Constraint checks are not deferred at all
2. DEFERRABLE INITIALLY DEFERRED: Constraint checks are deferred right at the start
3. DEFERRABLE INITIALLY IMMEDIATE: Constraint checks are not deferred until DEFERRED keyword in transaction (Defer constraints on demand)

2.4 Set Operations

- **Union Compatible:** 2 relations are union-compatible if (1): Both have same # of attributes, (2): Corresponding attributes have same or compatible domains (Similar to function signatures where number, order & type matters)
- **Remove Duplicates:** `UNION, INTERSECT, EXCEPT`
- **Keep Duplicates:** `UNION ALL, INTERSECT ALL, EXCEPT ALL` (Treats each element as distinct element)

2.5 Subqueries

- Appears in `SELECT, FROM, WHERE`
- **IN/NOT IN**: Subquery must return exactly 1 column | If expression matches any subquery row → IN returns TRUE, NOT IN returns FALSE | If subquery evaluates to empty table → IN returns FALSE, NOT IN returns TRUE
- **EXISTS/NOT EXISTS**: Subquery may return any # of columns | If expression matches any subquery row → EXISTS returns TRUE, NOT EXISTS returns FALSE | If subquery evaluates to empty table, EXISTS returns FALSE, NOT EXISTS returns TRUE | Only emptiness of subquery matters (Use `SELECT 1`)
- **ANY/ALL**: Subquery must return exactly 1 column | ANY returns TRUE if comparison is true to ≥ 1 row, ALL returns TRUE if comparison is true to all rows | If subquery evaluates to empty table, ANY returns FALSE, ALL returns TRUE

2.6 Conceptual Evaluation

1. **FROM**: Compute cross product/JOINs of all Tables in FROM clause
2. **WHERE**: Keep tuples that evaluates to TRUE on the WHERE condition (WHERE follows Principle of Acceptance | Cannot use aggregate directly in WHERE, but can have sub-query which contains aggregate in WHERE)
3. **GROUP BY**: Partition table into groups w.r.t grouping attributes (Application of aggregate functions are over each group → 1 result tuple per group)
4. **HAVING**: Keep groups that evaluates to TRUE on the HAVING condition (Conditions typically involve aggregates)
5. **SELECT**: Remove all attributes not specified in SELECT clause (Remove duplicates if DISTINCT)
6. **ORDER BY**: Sort tables based on specified attributes
7. **LIMIT/OFFSET**: Keep tuples based on their order in table

Restriction to SELECT/HAVING clauses upon GROUP BY: If column A_i of table R appears in SELECT/HAVING clause, A_i must appear in GROUP BY clause OR A_i appear as input of aggregate function in SELECT/HAVING clause OR PK of R appears in GROUP BY clause

2.7 Special Functions

1. **Patter Matching:** _ matches any single character, % matches any sequence of 0 or more characters (eg. `WHERE pizza LIKE 'Ma%a'`)
2. **COALESCE**: Returns 1st non-NULL value in list of input arguments, returns NULL if all values in list are NULL
3. **NULLIF:** Returns NULL if $value_1 = value_2$, otherwise return $value_1$

2.8 Universality

- **Double Negation Method:** $\forall x : Exist(x) == \sim \exists x : \sim Exist(x)$
- Eg. Restaurants that sells ALL pizzas liked by 'Homer' == There does not exist pizza that 'Homer' likes & not sold by the restaurant

```sql
SELECT DISTINCT S1.rname FROM Sells S1
WHERE  NOT EXISTS (
    SELECT 1 FROM Likes L
    WHERE  L.cname = 'Homer'
      AND  NOT EXISTS (
         SELECT 1 FROM Sells S
         WHERE  S.pizza = L.pizza
           AND  S.rname = S1.rname
      )
);
```

- **Cardinality Method:** $S \subseteq R \rightarrow |R \cup S| = |R|, |R \cap S| = |S|$
  $R \equiv S \rightarrow |R \cup S| = |R \cap S|$

2.9 Recursive CTE

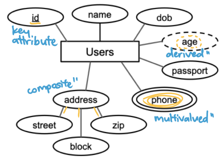
Eg. Find all MRT stations that can be reached from NS1 in at most 3 stops

```sql
WITH RECURSIVE
  Linker(to_stn, stops) AS (
    SELECT to_stn, 0 FROM MRT
    WHERE  fr_stn = 'NS1'
    UNION ALL
    SELECT M.to_stn, L.stops + 1
    FROM   Linker L, MRT M
    WHERE  L.to_stn = M.fr_stn
  )
SELECT DISTINCT (to_stn)
FROM   Linker WHERE stops < 3;
```

3 Entity Relationship Model

3.1 Entity Relationship

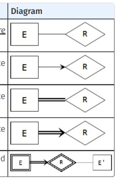
- **Entities:** Nouns (Rectangles), **Relationships:** Verbs (Diamonds)
- **Attributes:** Describe info about entities & relationships
  1. Key attribute: Uniquely identifies each Entity
  2. Composite attribute: Composed of multiple other attributes
  3. Multi-valued attributes: 1 or more values for a given Entity
  4. Derived attributes: Derived from other attributes



- **Degrees of Relationship Sets:** # of entity sets (can be non-unique) involved in relationship set

3.2 Relationship Constraints

Name	Constraint	Diagram
Unconstrained	Each instance of E may participate in 0 or more instance of R	
Key Constraint	Each instance of E participates in at most 1 instance of R	
Total Participation	Each instance of E participates in at least 1 instance of R	
Key + Total Participation	Each instance of E participates in exactly 1 instance of R	
Weak Entity + Identifying Relationship	E is a weak entity set with identifying owner E' and indentifying relationship set R	



Weak Entity Set: An entity set that does not have its own key (Has partial key that cannot uniquely identify an entity) → Needs help of key from Owning Entity Set to uniquely identify an entity

3.3 Relational Mapping (ER to Schema)

Entity Set

1. Name of entity set → Name of table
2. Attribute of entity set → Column of table
3. Key attribute of entity set → PK of table
4. Derived attribute of entity set → Should not appear in table
5. Composite attribute of entity set → Converted into decomposed attributes in table
6. Multi-valued attribute of entity set → Converted into seq. of single-valued attributes OR Create another table with FK constraint

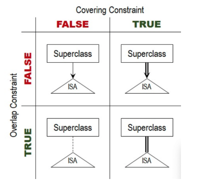
Relationship Set

1. Many-to-Many Relationship: Relationship Set Schema should have its PK including PKs of both entities
2. Many-to-One Relationship:
   - *Separate Tables strategy:* Relationship Set Schema should have PK of One-Set as its PK to enforce upper bound of 1 & PK of Many-Set in Relationship Set should be NOT NULL to prevent redundant info
   - *Combined Tables strategy:* Combine Relationship Set with One-Set, PK of merged set is PK of One-Set, PK of Many-Set in merged set can be NULL
3. One-to-One Relationship:
   - *No Relationship Set strategy:* PK of each entity set is a UNIQUE FK in other set
   - *3 Table strategy:* Relationship Set Schema has its PK as PK of 1 of the entity set, other entity set's PK appears in Relationship Set Schema as a UNIQUE, NOT NULL FK (candidate key)
4. Key + Total Relationship: Combine Relationship Set with KeyTotal-Set, PK of merged set is PK of KeyTotal-Set, PK of Many-Set in Relationship Set must be NOT NULL
5. Weak Entity Set Relationship: Combine Relationship Set with Weak Entity Set, PK of merged set is combination of Owning Entity Set's PK & Weak Entity Set's PK, PK of Owning Entity Set appears as FK in merged set with ON DELETE/UPDATE CASCADE

NOTE: Total Participation Constraint in ER model cannot be enforced by schema

3.4 ISA Hierarchy

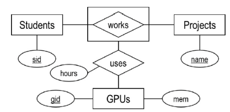
- Subclass must be uniquely identified by same key attributes of superclass (subclass keys should not be shown in ER diagram)
- Subclass may have additional attribute & involved in additional relationship
- **Schema Syntax:** Subclass has Superclass's PK as its PK & FK (along with ON DELETE CASCADE)



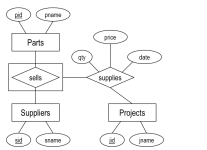
1. Overlap Constraint (Upper bound): Can a superclass belong to multiple subclasses? FALSE sets Upper bound = 1 (Key constraint)
2. Covering Constraint (Lower bound): Must a superclass belong to ≥ 1 subclass? TRUE sets Lower bound = 1 (Total Participation constraint)

3.5 Aggregation

- Aggregate is a relationship set & entity set
- Key attributes are formed from relationship set




```sql
CREATE TABLE Uses (
  gid   INT REFERENCES GPUs,
  sid   INT,
  name  VARCHAR(50),
  hours NUMERIC,  -- can have decimal point
  PRIMARY KEY (gid, sid, pname),
  FOREIGN KEY (sid, pname) REFERENCES Works (sid, pname)
); -- Foreign key to "Works"
  -- treating "Works" like an entity set
```



- Supplier can sell Part without being used by a Project
- Aggregate's relationship set depicts necessary relationship btw. 2 entity sets
- Aggregate's entity set depicts optional relationship

4 Relational Algebra

4.1 Algebra

Closure: A set of values is closed under the set of operators if any combination of the operators produces only values in the given set (Relations are closed under Relational Algebra)

4.2 Unary Operators

Selection Operator (σ)

- $\sigma_{[c]}(R)$: Selects all tuples from relation R (ROWS) that satisfies the selection condition c based on Principle of Acceptance
- c is a condition that returns a boolean (potentially NULL), c must specify only attributes in R
- Precedence: $(), op, \neg, \wedge, \vee$
- Can be mapped to `WHERE` clause of SQL
- Properties: Result have same schema as input relation | # of rows are often smaller

Projection Operator (π)

- $\pi_{[l]}(R)$: Keeps only columns specified in ordered list l and in same order
- l must specify only attributes in R, No operations & duplicates in projection operator (eg. $\pi_{[A_1+A_2]}(R)$, $\pi_{[A_1,A_1]}(R)$)
- $\pi_{[A_1,A_2]}(R) \neq \pi_{[A_2,A_1]}(R)$ (Order matters!!)
- Can be mapped to `SELECT` clause in SQL
- Properties: Resulting schema is as specified by l without relation name | # of rows may be smaller (Relation is defined as set of tuples → Duplicate rows will be removed)

Renaming Operator (ρ)

- $\rho_{[r]}(R)$: Renames all attributes mentioned in r s.t. for each renaming $B_i \leftarrow A_i$, A_i is renamed to B_i
- Can be mapped to `AS` keyword in `SELECT` in SQL
- Properties: Resulting schema is old schema renamed by r | Order of column is unchanged (except for renaming) | # of rows remains the same

4.3 Binary Operators

SET Operators

- $R \cap S$, $R \cup S$, $R - S$ (Relations must be union-compatible)

PRODUCT Operators

- $R \times S$: Cross product of 2 relations is a relation formed by combining all pairs of tuples from 2 input relations
- Can be mapped to `FROM` keyword in SQL
- Optimisation: Cross product can be optimised by `JOIN` operators, which combines cross product, selection & projection (Avoids generating all $|R \times S|$ intermediate tuples)

INNER JOIN Operators

1. θ **Join:** $R \bowtie_{[\theta]} S = \sigma_{[\theta]}(R \times S)$ (Cross product followed by selection)
2. **Equi Join**: $R \bowtie_{[=]} S$ = Special θ Join where the only relational operator used is equality (All equi join is a θ join but not all θ join are equi join)
3. **Natural Join:** $R \bowtie S = \pi_{[l]}(\sigma_{[\theta]}(R \times S))$
   - $l = (Attr(R) \cap Attr(S)) + (Attr(R) - Attr(S)) + (Attr(S) - Attr(R))$ (Order matters, where common attributes are listed, then R's attributes, followed by S's attributes)
   - $\theta = \forall \in (Attr(R) \cap Attr(S)) : R.A_i = S.A_i$ (All common attributes are equal)

OUTER JOIN Operators

1. **Semi Join:** $R \ltimes_{[\theta]} S = \pi_{[Attr(R)]}(R \bowtie_{[\theta]} S)$ (Used to find non-dangling tuple)
   NOTE: Semi Join projects R's attributes only, while normal Joins will project R and S's attributes
2. **Dangles:** $dangle(R \bowtie_{[\theta]} S) = R - R \ltimes_{[\theta]} S$
3. **Left Outer Join:** $R_1 ⟕_{[c]} R_2$ = Inner Join ∪ Left Dangling Tuple = $R_1 \bowtie_{[c]} R_2 \cup (dangle(R_1 \bowtie_{[\theta]} R_2) \times (null(R_2))$
4. **Right Outer Join:** $R_1 ⟖_{[c]} R_2$ = Inner Join ∪ Right Dangling Tuple = $R_1 \bowtie_{[c]} R_2 \cup ((null(R_1) \times dangle(R_2 \bowtie_{[\theta]} R_1))$
5. **Full Outer Join:** $R_1 ⟗_{[c]} R_2$ = Inner Join ∪ Left Dangling Tuple ∪ Right Dangling Tuple = $R_1 \bowtie_{[c]} R_2 \cup (dangle(R_1 \bowtie_{[\theta]} R_2) \times (null(R_2)) \cup ((null(R_1) \times dangle(R_2 \bowtie_{[\theta]} R_1))$

4.4 Complex Expressions

Equivalence VS Isomorphic

- Equivalence ($\equiv$): 2 relational algebra expressions are equivalent if both produces same result with same column order, possibly different row order
- Isomorphic ($\cong$): 2 relational algebra expressions are isomorphic if both produces same result with possibly different column order, possibly different row order

Special Properties

1. $\sigma_{[\theta_1]}(\sigma_{[\theta_2]}(R)) \equiv \sigma_{[\theta_2]}(\sigma_{[\theta_1]}(R)) \equiv \sigma_{[\theta_1 \wedge \theta_2]}(R)$
2. $\pi_{[l_1]}(\pi_{[l_2]}(R)) \not\equiv \pi_{l_1}(R)$ unless $l_1 \subseteq l_2$
3. $R \times S \cong S \times R$, $R \bowtie S \cong S \bowtie R$, $R \bowtie (S \bowtie T) \cong (R \bowtie S) \bowtie T$ (Different column order)
4. $R \times (S \times T) \equiv (R \times S) \times T$ (Associative)
5. $\pi_{[l]}(\sigma_{[\theta]}(R)) \not\equiv \sigma_{[\theta]}(\pi_{[l]}(R))$ (Unless θ uses only attributes in l)
6. $\sigma_{[\theta]}(R \times S) \not\equiv \sigma_{[\theta]}(R) \times S$ (Unless θ uses only Attr(R))

5 Functions & Procedures

5.1 Functions

- Returns some values/tuples
- **Returns output of an atomic data type:**

```
CREATE OR REPLACE FUNCTION convert (mark INT)
RETURNS char(1) AS $$
        SELECT  CASE
                WHEN mark >=70 THEN 'A'
                WHEN mark >= 60 THEN 'B'
                WHEN mark >= 50 THEN 'C'
                ELSE   'D'
                END;
$$ LANGUAGE sql;
```

1 letter output

Example Query:

`SELECT Name, convert(Mark) FROM Scores;`

- **Returns 1 EXISTING tuple:**

```
CREATE OR REPLACE FUNCTION topStudent ()
RETURNS Scores AS $$
        SELECT      *
        FROM        Scores
        ORDER BY    Mark  DESC LIMIT 1;
$$ LANGUAGE sql;
```

Example Query: `SELECT topStudent();`

- **Returns EXISTING SET of tuples:**

```
CREATE OR REPLACE FUNCTION topStudents ()
RETURNS SETOF Scores AS $$
        SELECT      *
        FROM        Scores
        WHERE       Mark =
                (SELECT MAX(Mark) FROM Scores);
$$ LANGUAGE sql;
```

* Cannot directly put aggregate function in WHERE (only SELECT & HAVING)
↳ ∴ cannot put WHERE Mark = MAX(Mark)

Example Query: `SELECT * FROM topStudents();`

- **Returns 1 NEW tuple (Note parameters):**

```
CREATE OR REPLACE FUNCTION topMarkCnt (OUT TopMark INT,
        OUT Cnt INT)
RETURNS RECORD AS $$
        SELECT Mark, COUNT(*)
        FROM   Scores
        WHERE Mark = (SELECT MAX(Mark) FROM Scores)
        GROUP BY      Mark ;
$$ LANGUAGE sql;
```

Can omit this part

Example Query: `SELECT topMarkCnt();`

- **Returns NEW SET of tuples (Note parameters):**

```
CREATE OR REPLACE FUNCTION MarkCnt (OUT Mark INT,
        OUT Cnt INT)
RETURNS SETOF RECORD AS $$
        SELECT Mark, COUNT(*)
        FROM  Scores
        GROUP BY    Mark ;
$$ LANGUAGE sql;
```

```
CREATE OR REPLACE FUNCTION MarkCnt ()
RETURNS TABLE(Mark INT, Cnt INT) AS $$
        SELECT Mark, COUNT(*)
        FROM   Scores
        GROUP BY    Mark ;
$$ LANGUAGE sql;
```

Example Query: `SELECT MarkCnt();`

5.2 Procedures

- Do not return value/tuple, treated as a transaction
- Syntax: `CREATE OR REPLACE PROCEDURE...`

5.3 Control Structures:

1. `IF ... THEN ... ELSE ... END IF`
2. `LOOP ... END LOOP`
3. `EXIT ... WHEN ...`
4. `WHILE ... LOOP ... END LOOP`
5. `FOR ... IN ... LOOP ... END LOOP`

5.4 Variables:

- Variables are declared in `DECLARE` section, Actual function is in `BEGIN ... END` section
- Values are selected into variables (`SELECT COUNT(*) INTO overlap_count`) OR assigned directly (`temp_val := val1`)

5.5 Cursor:

- Cursor enables us to access each individual row selected by a `SELECT` statement (DECLARE → OPEN → FETCH → CLOSE)

```
CREATE OR REPLACE FUNCTION score_gap()
RETURNS TABLE ( name TEXT, mark INT, gap INT ) AS $$
DECLARE
        curs CURSOR FOR (SELECT * FROM Scores ORDER BY Mark DESC);
        r RECORD;
        prv_mark INT;
BEGIN
        prv_mark := -1;
        OPEN curs;
        LOOP
                FETCH curs INTO r;
                EXIT WHEN NOT FOUND;
                name := r.Name;
                mark := r.Mark;
                IF prv_mark >= 0 THEN gap := prv_mark - mark;
                ELSE gap := NULL;
                END IF;
                RETURN NEXT;
                prv_mark := r.Mark;
        END LOOP;
        CLOSE curs;
END;
$$ LANGUAGE plpgsql;
```

After last row.

- `OPEN`: SQL statement for cursor is executed & cursor points to beginning of result
- `FETCH`: Next tuple from cursor is read & put into `r` → No tuple, terminate loop
- `RETURN NEXT`: Inserts a tuple to output of function
- `CLOSE`: Releases resources allocated to cursor
- Cursor Movement: `FETCH PRIOR FROM cur INTO r`, `FETCH FIRST FROM cur INTO r`, `FETCH LAST FROM cur INTO r`, `FETCH ABSOLUTE x FROM cur INTO r` (fetch x^{th} tuple)

6 Triggers

6.1 Basics of Triggers & Trigger Functions

Triggers: Condition that database has to check whenever appropriate

```
CREATE TRIGGER scores_log_trigger
AFTER INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION scores_log_func();
```

- Tells database to watch out for insertions on Scores → Calls `scores_log_func()` after each insertion of a tuple

Trigger Functions: Expression of the condition about something done to database

```
CREATE OR REPLACE FUNCTION scores_log_func() RETURNS TRIGGER
AS $$
BEGIN
        INSERT INTO Scores_Log(Name, EntryDate)
                VALUES (NEW.Name, CURRENT_DATE);
        RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

from the new tuple inserted.

- `RETURNS TRIGGER` indicates that this is a trigger function (Only TRIGGER functions have access to `NEW` keyword)
- Trigger functions should not have any input parameters
- Trigger functions can access
  1. `TG_OP`: Operation that activated trigger (INSERT, UPDATE, DELETE)
  2. `TG_TABLE_NAME`: Name of table that activated trigger
  3. `OLD`: Old tuple being updated/deleted
  4. `NEW`: New tuple to update/ to be inserted

6.2 Trigger Operations

1. **Insert:** OLD (NULL tuple) | NEW (Non-NULL tuple)
2. **Update:** OLD (Non-NULL tuple) | NEW (Non-NULL tuple)
3. **Delete:** OLD (Non-NULL tuple) | NEW (NULL tuple)

6.3 Trigger Timing

1. **AFTER:** Function would be executed AFTER tuple operation
   - AFTER INSERT: Return value does not matter
   - AFTER UPDATE: Return value does not matter
   - AFTER DELETE: Return value does not matter
   - Reason: Trigger function is invoked after main operation is done
2. **BEFORE:** Function would be executed BEFORE tuple operation
   - BEFORE INSERT: Non-NULL tuple returned → tuple will be inserted | NULL tuple returned → no tuple inserted
   - BEFORE UPDATE: Non-NULL tuple returned → tuple will be updated tuple | NULL tuple returned → no tuple updated
   - BEFORE DELETE: Non-NULL tuple returned → deletion proceeds as normal | NULL tuple returned → no deletion
   - `RETURN NULL`: Tells database to ignore rest of operation
3. **INSTEAD OF:** Function would be executed INSTEAD OF tuple operation (Can be defined on views only, instead of doing something on a view, do it on a table)
   - Returning NULL: Signals database to ignore rest of operation on current row
   - Returning non-NULL tuple: Signals database to proceed as normal

6.4 Trigger Levels

- **FOR EACH ROW**: Row-level trigger that executes trigger function for every tuple encountered
- **FOR EACH STATEMENT**: Statement-level trigger that executes trigger function only once
  - Statement-level triggers ignore values returned by trigger operations, `RETURN NULL` will not make database omit subsequent operations → Need to RAISE EXCEPTION instead of RAISE NOTICE to prevent deletion
- `INSTEAD OF` only allowed on row-level
- `BEFORE/AFTER` allowed on both row-level & statement-level

6.5 Trigger Condition

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW
WHEN (NEW.Name = 'Elise')
EXECUTE FUNCTION for_Elise_func();
```

Trigger function will only be activated when this boolean condition is True

- No `SELECT` in `WHEN()`
- No OLD in `WHEN()` for `INSERT`
- No NEW in `WHEN()` for `DELETE`
- No `WHEN()` for INSTEAD OF

6.6 Deferred Trigger

- Defers checking of triggers
- **Syntax:**
  - `CONSTRAINT` and `DEFERRABLE` together → Trigger can be deferred
  - `INITIALLY DEFERRED` by default → Trigger is deferred
  - `INITIALLY IMMEDIATE` → Trigger is not deferred by default
  - Deferred triggers only works with AFTER (to defer trigger, has to execute after main operation) & FOR EACH ROW
- **Procedure:**
  1. Put inter-dependent statements into 1 transaction
  2. Defer trigger check to end of transaction (Trigger is activated at COMMIT)

```
CREATE CONSTRAINT TRIGGER bal_check_trigger
AFTER INSERT OR UPDATE OR DELETE ON Account
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION bal_check_func();

BEGIN TRANSACTION;
UPDATE Account SET Bal = Bal – 100 WHERE AID = 1;
UPDATE Account SET Bal = Bal + 100 WHERE AID = 2;
COMMIT;
```

```
CREATE CONSTRAINT TRIGGER bal_check_trigger
AFTER INSERT OR UPDATE OR DELETE ON Account
DEFERRABLE INITIALLY IMMEDIATE
FOR EACH ROW
EXECUTE FUNCTION bal_check_func();

BEGIN TRANSACTION;
SET CONSTRAINTS bal_check_trigger DEFERRED;
UPDATE Account SET Bal = Bal – 100 WHERE AID = 1;
UPDATE Account SET Bal = Bal + 100 WHERE AID = 2;
COMMIT;
```

6.7 Multiple Triggers

- **Order of trigger activation:**
  1. BEFORE statement-level triggers
  2. BEFORE row-level triggers
  3. AFTER row-level triggers
  4. AFTER statement-level triggers
- Within each category, triggers are activated in alphabetical order
- If BEFORE row-level trigger returns NULL → subsequent triggers on same row are omitted

7 Functional Dependencies

7.1 Functional Dependency

- **Definition:** If attribute A uniquely decides attribute B → there is a FD from A to B ($A \to B$)
- **Formal Definition:** $A_1A_2...A_n \to B_1B_2...B_n$ if whenever 2 objects have the same values on $A_1A_2...A_n$, they always have same values on $B_1B_2...B_n$
- **FDs on Tables:** An FD may hold on 1 table but does not hold on another
- **Techniques to spot FDs:**
  1. Given $A \to B$ & A is FALSE, the FD is vacuously TRUE (A is FALSE means that there are no ≥ 2 tuples having same A values)
  2. Come up with a counter example to the requirement & put column with same values on the LHS, column with different values on the RHS (Eg. No 2 customers buy the same product: C_1, P_1 & C_2, P_1 violates requirement → Place P on LHS, place C on RHS ($P \to C$))

7.2 Armstrong Axioms & Rules

1. **Reflexivity:** $AB \to A$
2. **Augmentation:** If $A \to B$ then $AC \to BC$
3. **Transitivity:** If $A \to B$ and $B \to C$ then $A \to C$
4. **Decomposition:** If $A \to BC$ then $A \to B$ and $A \to C$
5. **Union:** If $A \to B$ and $A \to C$ then $A \to BC$

7.3 Closure

- **Definition:** $\{A_1, A_2, ..., A_n\}^+$ = Set of attributes that can be decided by $A_1, A_2, ..., A_n$ directly or indirectly
- **Computing Closures:**
  1. Initialise closure to $\{A_1, A_2, ..., A_n\}$
  2. If there is an FD: $A_i, A_j, ..., A_m \to B$, such that $A_i, A_j, ..., A_m$ are all in closure, then put B into closure
  3. Repeat step 2 until we cannot find any new attribute to put into closure
- **Using Closures to prove FDs:**
  - To prove that $A \to B$ holds, only need to show that $\{A\}^+$ contains B
  - To prove that $A \to B$ does not hold, only need to show that $\{A\}^+$ does not contain B

7.4 Keys, Superkeys, Prime Attributes

- **Superkeys of a table:** Set of attributes in a table that decides all other attributes
- **Keys of a table:** A superkey that is minimal (If we remove any attribute from superkey, it will not be a superkey anymore | A table may have multiple keys)
- **Prime Attribute:** If an attribute appears in a key, then it is a prime attribute

Algorithm for finding keys: Given table $T(A, B, C...)$ and a set of FDs on T

1. Consider every subset of attributes in T
2. Derive the closure of each subset
3. Identify all superkeys based on closures (Pick closures that contain all attributes of T)
4. Identify all keys from superkeys

   Tricks: (1): Check all small attribute sets first (If closure of the set contains all attributes, no need to check superset already) | (2): If an attribute does not appear on RHS of any FD, then it must be in every key

8 BCNF

8.1 Non-trivial & Decomposed FD

- **Decomposed FD:** An FD whose RHS has only 1 attribute (A non-decomposed FD can always be transformed into a set of decomposed FDs via Rule of Decomposition)
- **Types of FDs:**
  - $\alpha \to \beta$ is a ***trivial*** functional dependency if $\beta \subseteq \alpha$
  - $\alpha \to \beta$ is a ***non-trivial*** functional dependency if $\beta \not\subseteq \alpha$
  - $\alpha \to \beta$ is a ***completely non-trivial*** functional dependency if $\alpha \cap \beta = \emptyset$
- **Non-trivial & Decomposed FD** A decomposed FD whose RHS does not appear in LHS
- **Algorithm for finding Non-trivial & decomposed FDs (via Closures)**: Consider $R(A, B, C)$
  1. Consider all attribute subsets in R
  2. Compute closure of each subset
  3. From each closure, remove trivial attributes (remove attributes from closure that appear in original subset)
  4. Derive non-trivial & decomposed FDs from each closure (use Rule of Decomposition if needed)

8.2 BCNF

- **Definition:** Table R is in BCNF if every non-trivial & decomposed FD has a superkey in its LHS
- **Algorithm to check BCNF:**
  1. Compute closure of each attribute subset
  2. Derive keys of R (using closure)
  3. Derive all non-trivial & decomposed FDs on R (using closure)
  4. Check if all non-trivial & decomposed FDs' LHS is a superkey → if all non-trivial & decomposed FDs satisfy requirement, R is in BCNF
- **Simplified Algorithm to check BCNF:**
  1. Compute closure of each attribute subset
  2. Check if there is a closure such that it satisfies "more than but not all" condition
  3. If such a closure exists → R is NOT in BCNF
- **Properties of BCNF:**
  1. No update or deletion or insertion anomalies
  2. Small redundancy
  3. Original table can always be reconstructed from decomposed tables
  4. Dependencies may not be preserved in decomposed table

8.3 BCNF Decomposition

If table is not in BCNF → Decompose table into smaller tables (Normalisation)

- **Decomposition Algorithm:**
  1. Find a subset X of attributes in R such that its closure satisfies "more but not all" condition
  2. Decompose R into 2 tables R_1, R_2 such that R_1 contains all attributes in $\{X\}^+$ & R_2 contains all attributes in X as well as attributes not in $\{X\}^+$
  3. If R_1 is not in BCNF → decompose R_1, If R_2 is not in BCNF → decompose R_2

  NOTE: BCNF decomposition of a table may not be unique | Table only has 2 attributes → Table MUST be in BCNF
- **Projection of Closures/FDs:** Used when we want to derive closures on a table R_i that is decomposed from a table R → Can decide whether R_i is in BCNF & whether to decompose R_i
  1. Enumerate attribute subsets of R_i
  2. For each subset, derive its closure on R (basically use FDs from R)
  3. Project each closure onto R_i by removing those attributes that do not appear in R_i
- **Lossless Join Decomposition**
  - Decomposition guarantees lossless join whenever common attributes in R_1 & R_2 is a superkey of R_1 or R_2
  - WORKING: For LJD with multiple decomposed schema, working only requires 1 possible decomposition & show the working that each step is a LJD | For non-LJD, working requires exploration of all possibilities & explain why each possibility is non-LJD
  - BCNF guarantees lossless join

9 3NF

9.1 Dependency Preservation

- Let S be given set of FDs on original table, S' be set of FDs on decomposed table
- Decomposition preserves all FDs ↔ S & S' are equivalent (Every FD in S' can be derived from S, Every FD in S can be derived from S')

9.2 3rd Normal Form

- **Definition:** A table satisfies 3NF ↔ For every non-trivial & decomposed FD either (1) LHS is a superkey or (2) RHS is a prime attribute (appears in a key)
- **Properties of 3NF:**
  1. Not as strict as BCNF
  2. Small redundancy (not as small as BCNF)
  3. Lossless join property
  4. Preserves all FDs
- **BCNF VS 3NF:** Satisfying BCNF → Satisfying 3NF (but not necessarily vice versa) | Violating 3NF → Violating BCNF (but not necessarily vice versa)
- **Algorithm to check 3NF:**
  1. Derive keys of R (via Algorithm to find keys)
  2. For each given FD, check if LHS is a superkey OR Each attribute on RHS is a prime attribute
  3. If all given FDs satisfy this condition → R is in 3NF

9.3 3NF Decomposition

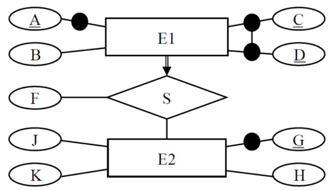
A table is NOT in 3NF → Decompose it into smaller tables that are in 3NF

- **BCNF Decomposition VS 3NF Decomposition:** BCNF Decomposition may perform ≥ 1 binary splits, each of which divides a table into 2 | 3NF decomposition has only 1 split, which divides table into ≥ 2 parts
- **3NF Decomposition Algorithm:**
  1. Given a table R and a set S of FDs, derive minimal basis of S
  2. In minimal basis, combine FDs whose LHSs are the same (Basically get non-decomposed FDs aka. Canonical minimal basis)
  3. Create a table for each FD remained
  4. If none of the tables contains a key of original table R, create a table that contains any key of R (Ensure lossless join decomposition)
  5. Remove subsumed tables (Remove table if all of its attributes are contained in another table)

9.4 Minimal Basis

- Minimal Basis of S is a simplified version of S
- **Conditions:**
  1. Every FD in minimal basis can be derived from S, and vice versa
  2. Every FD in minimal basis is a non-trivial & decomposed FD
  3. No FD in minimal basis is redundant (No FD in minimal basis can be derived from other Fds in minimal basis)
  4. For each FD in minimal basis, none of attributes on LHS is redundant (If we remove an attribute from LHS, then resulting FD is a new FD that cannot be derived from original set of FDs)
- **Algorithm to find Minimal Basis**
  1. Transform FDs, so that each RHS contains only 1 attribute (Decompose FDs)
  2. Remove redundant attributes on LHS of each FD
     a. Given $AB \to C$, remove A to produce $B \to C$
     b. Check whether $B \to C$ is implied by S (Check if closure of B contains C)
     c. If $B \to C$ is not implied, A is NOT redundant, If $B \to C$ is implied, A is redundant
  3. Remove redundant FDs (Remove FD from S & see if that FD can be derived from other FDs in S)

9.5 Primary Keys & FDs



- **Candidate Keys:** A, CD for E1 | G for E2
- **Functional Dependencies:** A → CD, CD → A, A → B, CD → B, A → F, CD → F, A → G, CD → G, G → HJK